

# Още задачи върху динамично програмиране

Тодор Дуков

## Два интересни примера

За първия пример, нека имаме някакъв краен речник  $D$  от непразни думи над  $\Sigma = \{a, \dots, z\}$ . Ще искаме при подаден низ над  $\Sigma$  да видим дали той може да се разбие на думи от речника (с възможни повторения). Нека например да вземем речника  $D = \{\text{mango}, \text{i}, \text{icescream}, \text{like}, \text{with}\}$ . Тогава думата “ilikeicescreamwithmango” може да се разбие на “i like icescream with mango”.

Нека е подаден един низ  $\alpha \in \Sigma^*$ . Ако  $\alpha = \varepsilon$ , то тогава очевидно можем да получим  $\alpha$  чрез конкатенация на думи от  $D$ . Ако  $\alpha \neq \varepsilon$  и  $\alpha$  се получава чрез конкатенация на думи от  $D$ , то тогава има  $\beta \in \Sigma^*$  и  $\gamma \in D$ , за които е изпълнено, че  $\alpha = \beta\gamma$  и  $\beta$  може да се получи чрез думи от  $D$ . Но  $\gamma \neq \varepsilon$ , т.е. успяхме да сведем задачата за  $\alpha$  до задача за  $\beta$ , като  $|\beta| < |\alpha|$ . Нека сега да формализираме тези разсъждения. Нека  $\alpha = \alpha_1 \dots \alpha_n$ , където  $\alpha_i \in \Sigma$ . Тогава булевата функция  $\text{WB}_{D,\alpha}(i)$ , която казва дали  $\alpha_1 \dots \alpha_i$  може да се разбие на думи от  $D$ , изглежда така:

$$\text{WB}_{D,\alpha}(i) = \begin{cases} \mathbb{T} & , \text{ ако } i = 0 \\ \bigvee_{j=1}^i (\alpha_j \dots \alpha_i \in D \ \& \ \text{WB}_{D,\alpha}(j-1)) & , \text{ иначе} \end{cases}$$

На нас това, което ни трябва, е  $\text{WB}_{D,\alpha}(|\alpha|)$ . С малко мислене върху това как се пресмята  $\text{WB}_{D,\alpha}$ , човек може да стигне до следното итеративно решение:

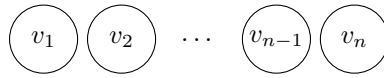
```
1 bool word_break(string s, vector<string> &word_dict)
2 {
3     int n = s.size();
4     vector<bool> wb(n + 1, false);
5     wb[0] = true;
6
7     for (int i = 1; i <= n; ++i)
8     {
9         for (const string &word : word_dict)
10        {
11            if (i < word.size())
12                continue;
13
14            if (i == word.size() || wb[i - word.size()])
15            {
16                if (s.substr(i - word.size() + 1, word.size()) == word)
17                {
18                    wb[i] = true;
19                    break;
20                }
21            }
22        }
23    }
24
25    return wb[n];
26 }
```

Ако  $|\alpha| = n$ ,  $|D| = m$ , и  $\max\{|\beta| \mid \beta \in D\} = k$ , то това решение има сложност по време  $O(n \cdot m \cdot k)$ , понеже за всяко  $1 \leq i \leq n$  и за всяка дума  $\beta \in D$  (те са  $m$  на брой) проверяваме дали  $\alpha_{i-|\beta|+1} \dots \alpha_i = \beta$ , което става за време  $O(k)$ , и дали  $\text{WB}_{D,\alpha}(i - |\beta|) = \mathbb{T}$ , което поради наличието на масива **wb** става за константно време. Този масив доста помага, ако не го бяхме ползвали, сложността щеше да се опише (горе долу) с рекурентното уравнение:

$$T(n) = \sum_{i=1}^n (m \cdot k \cdot T(n-i)) + \Theta(1).$$

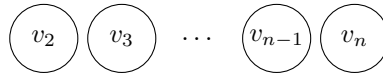
Сложността по памет очевидно е  $\Theta(n)$ , заради допълнителния масив, който заделяме.

Вторият пример ще бъде под формата на игра. Наредени са  $n$  монети със стойности съответно  $v_1, \dots, v_n$ . Редуваме се с опонент да теглим една монета от избран от краищата на редицата, докато монетите не свършат. Накрая всеки човек печели толкова, колкото е изтеглил. В случай че играем първи, каква печалба можем да си гарантираме? Първо да започнем с двата най-прости типа игри – с една или две монети. В играта с една монета е ясно, че най-голямата печалба, която можем да си гарантираме, е стойността на монетата. В игра с две монети със стойности съответно  $v_1, v_2$ , най-голямата гарантирана печалба е очевидно  $\max\{v_1, v_2\}$ . Нека сега в общия случай имаме следната партия:



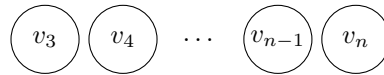
Ще се опитаме да сведем тази по-сложна партия, до няколко по-прости. Имаме две възможности:

1. Ако изберем първата монета, опонента ще трябва да избира в конфигурацията:



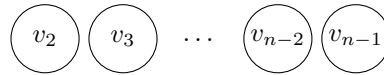
Той също има две възможности:

- (а) Ако опонента избере първата монета, ни оставя в конфигурацията:



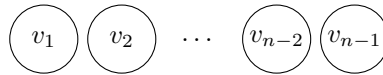
Тук можем да си мислим, че ние сме си заделили на страна печалба  $v_1$ , опонента си е заделил на страна печалба  $v_2$ , и играта започва наново в горната конфигурация.

- (б) Ако пък избере последната монета, ни оставя в конфигурацията:



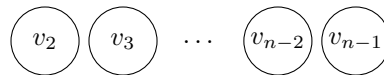
Тук можем да си мислим, че ние сме си заделили на страна печалба  $v_1$ , опонента си е заделил на страна печалба  $v_n$ , и играта започва наново в горната конфигурация.

2. Ако изберем последната монета, опонента ще трябва да избира в конфигурацията:



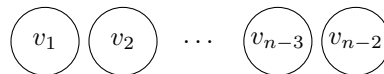
Той също има две възможности:

- (а) Ако опонента избере първата монета, ни оставя в конфигурацията:



Тук можем да си мислим, че ние сме си заделили на страна печалба  $v_n$ , опонента си е заделил на страна печалба  $v_1$ , и играта започва наново в горната конфигурация.

- (б) Ако пък избере последната монета, ни оставя в конфигурацията:



Тук можем да си мислим, че ние сме си заделили на страна печалба  $v_n$ , опонента си е заделил на страна печалба  $v_{n-1}$ , и играта започва наново в горната конфигурация.

Нека  $MP(i, j)$  е максималната печалба, която може да се спечели от първия играч (в първоначалната игра), ако може да събира монети със стойности съответно  $v_i, \dots, v_j$ . По предните разсъждения можем да пресметнем тази печалба рекурсивно така:

$$MP(i, j) = \begin{cases} v_i & , \text{ ако } i = j \\ \max\{v_i, v_j\} & , \text{ ако } i = j + 1 \\ \max\{v_i + \min\{MP(i + 2, j), MP(i + 1, j - 1)\}, v_j + \min\{MP(i + 1, j - 1), MP(i, j - 2)\}\} & , \text{ иначе} \end{cases}$$

Във хода на втория играч минимизираме, защото той печели възможно най-много, когато ние печелим възможно най-малко. За задача на читателя оставяме да пресметне  $MP(1, n)$  итеративно.

## Задачи

**Задача 1.** Формула ще наричаме всеки низ от вида  $B_0\sigma_1B_1\sigma_2B_2\ldots B_{n-1}\sigma_nB_n$ , където  $B_i \in \{\mathbb{T}, \mathbb{F}\}$  и  $\sigma_i \in \{\vee, \wedge, \oplus\}$ . Например низът  $\mathbb{T} \wedge \mathbb{T} \oplus \mathbb{F} \vee \mathbb{T}$  е формула. В зависимост от това как слагаме скобите, оценката на израза може да е различна. Например изразът  $(\mathbb{T} \wedge (\mathbb{T} \oplus (\mathbb{F} \vee \mathbb{T})))$  се остойностява като  $\mathbb{F}$ , докато изразът  $(\mathbb{T} \wedge ((\mathbb{T} \oplus \mathbb{F}) \vee \mathbb{T}))$  се остойностява като  $\mathbb{T}$ , въпреки че и двата израза се получават от примерната формула. Да се напише колкото се може по-бърз алгоритъм, който при подадена формула да върне броят на различни скобувания, за които съответния израз се остойностява като  $\mathbb{T}$ . След това да се докаже неговата коректност, и да се изследва сложността му по време.

**Задача 2.** Имаме професионален крадец, който иска да ограби къщите в дадена улица. Проблемът е, че ако той ограби две съседни къщи, алармата ще се активира и полицията ще дойде. Той не иска това, защото в такъв случай няма да спечели нищо. Да се напише колкото се може по-бърз алгоритъм, който при подаден масив от естествени числа  $L[1 \ldots n]$ , където  $L[i]$  е печалбата от къща  $i$ , връща максималната печалба на крадеца. След това да се докаже неговата коректност, и да се изследва сложността му по време.

**Задача 3.** Да се напише колкото се може по-бърз алгоритъм, който при подадена булева матрица  $M[1 \ldots n, 1 \ldots m]$  намира най-голямото квадратче в  $M$ , съставено от 1, и връща неговото лице. След това да се докаже неговата коректност, и да се изследва сложността му по време.

**Задача 4.** Един целочислен масив  $A[1 \ldots n]$  наричаме аритметичен, ако  $n \geq 3$  и за всяко  $1 \leq i \leq n - 2$  е изпълнено, че  $A[i+2] - A[i+1] = A[i+1] - A[i]$ . Да се напише колкото се може по-бърз алгоритъм, който при подаден целочислен масив  $A[1 \ldots n]$  намира броя на подредиците на  $A$ , които са аритметични масиви. След това да се докаже неговата коректност, и да се изследва сложността му по време.

**Задача 5.** Имаме  $n$  на брой къщи, които искаме да боядисаме със цветове  $c_1, c_2, c_3$ , като не може две съседни къщи да имат еднакъв цвят. Масив на цените ще наричаме всеки двумерен масив от положителни числа  $P[1 \ldots n, 1 \ldots 3]$ , където  $P[i, j]$  ще бъде цената за боядисване на къща  $i$  със цвят  $c_j$ . Да се напише колкото се може по-бърз алгоритъм, който при подаден двумерен масив от положителни числа, намира минималната цена за боядисване на всички къщи. След това да се докаже неговата коректност, и да се изследва сложността му по време.

**Задача 6.** Върху един масив от естествени числа  $A[1 \ldots n]$  можем да прилагаме следните две операции:

- да увеличим  $A[i]$  с единица за някое  $1 \leq i \leq n$ ;
- да намалим  $A[i]$  с единица за някое  $1 \leq i \leq n$ .

Да се напише колкото се може по-бърз алгоритъм, който при подаден масив от естествени числа  $A[1 \ldots n]$  връща минималния брой операции, които са нужни, за да стане масива монотонно ненамаляващ. След това да се докаже неговата коректност, и да се изследва сложността му по време.

**Задача 7.** Нека  $n, l, r \in \mathbb{N}$  и  $l \leq r$ . Един масив  $A[1 \ldots n]$  ще наричаме  $(n, l, r)$ -интересен, ако:

- $l \leq A[i] \leq r$  за всяко  $1 \leq i \leq n$ ;
- $\sum_{i=1}^n A[i] \equiv 0 \pmod{3}$ .

Да се напише колкото се може по-бърз алгоритъм, който при подадени  $n, l, r \in \mathbb{N}$ , за които  $l \leq r$ , връща броя на  $(n, l, r)$ -интересни масиви. След това да се докаже неговата коректност, и да се изследва сложността му по време.